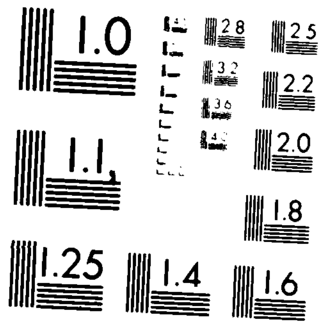
AD-A131 900 SOFTWARE VOTING IN ASYNCHRONOUS NMR (N-MODULAR REDUNDANCY) COMPUTER STRUC..(U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE G YORK ET AL. 1/1

UNCLASSIFIED 06 MAY 83 CMU-CS-83-128 DASG60-80-C-0057 F/G 9/2 NL

END
DATE
FILMED
9 83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

CMU-CS-83-128

2

AD A131

Software Voting in Asynchronous NMR Computer Structures

Gary York

Daniel Siewiorek

Zary Segall

6 May 1983

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

83 08 16 047

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-83-128	2. GOVT ACCESSION NO. AD. A131900	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Software Voting in Asynchronous NMR Computer Structures		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Gary York, Daniel Siewiorek and Zary Segall		8. CONTRACT OR GRANT NUMBER(s) ✓ DAAG-60-80-C-0057
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA. 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Marvin Denicoff, Program Director Information Systems Dept of Naval Research; ARlington, VA 22217		12. REPORT DATE May 6 1983
		13. NUMBER OF PAGES 34
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Information Systems Dept of the Navy Office of Naval Research Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Approved for public release; distribution unlimited		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

CMU CS 83 128

Software Voting in Asynchronous NMR Computer Structures

Gary York

Daniel Siewiorek

Zary Segall

6 May 1983

Department of Electrical Engineering
and Department of Computer Science
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

This research was supported by the Ballistic Missile Defense Advanced Technology Center under contract number DASG-60-80-C-0057, by the NSF under contract number ECS 8120270 and by NASA Langley Research Center under contract number NAG-1-190. The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the official policies, either expressed or implied, of NASA, the NSF, the BMDATC, or the U. S. Government.

Table of Contents

List of Figures

Software Voting in Asynchronous NMR Computer Structures

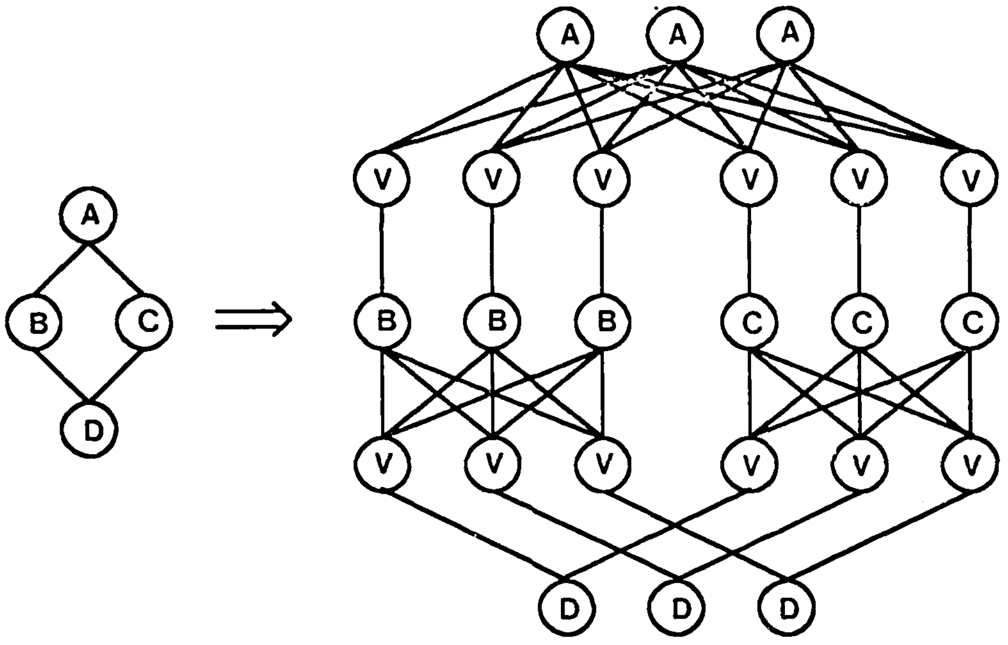
1. Introduction

Modern computer systems are being used in environments that require increased reliability due to the nature of the tasks being performed. For example, future avionics computers will replace the mechanical control of present aircraft. The computers will make thousands of decisions per second concerning the stability of the aircraft. The system must be designed so that the computer will never fail in flight, since the stability decisions can not be made by the pilot. In many cases, the required reliability is being obtained by replicating hardware components, and comparing the outputs of the components to determine the correct result. The replication allows the system to tolerate failures in components without affecting the system reliability.

One technique used to improve the system reliability is to replicate the hardware an odd number of times, and to compare the outputs of the modules to determine whether a majority of the modules agree. If a majority do agree, then this output is assumed to be the correct output. The comparison to determine a majority is called **voting**. The system that performs the comparison on the module outputs is called a **voter**. If the hardware is replicated three times, then the system has Triple Modular Redundancy (TMR). The generalization of TMR is N-Modular Redundancy (NMR), in which each module is replicated N times, and all N outputs are voted on to determine the correct output. Figure 1-1 shows a TMR system with four modules. The total number of modules and voters required for TMR is 3·(*number of modules* + *number of communication paths*).

Triple Modular Redundancy is a useful technique to mask failures in a system. One module can fail completely in a TMR system and the output of the system should not be affected. The number of redundant modules can be increased if the system reliability must be increased. The system reliability can therefore be increased by simply increasing the redundancy. The cost of this replication can be high. The ideal system performance of N processors is N times the performance of one processor. In a NMR system, though, all the processors are performing the same task, so the throughput is the same as for one processor. In fact, the performance will be worse than that of one processor because some overhead will be associated with the voting, thereby reducing the system throughput. The added reliability is exchanged for increased system cost and decreased throughput. Some applications require extremely reliable systems, so the only option is NMR. Many modern computer systems use some type of replication to increase reliability.

In N-modular-redundancy (NMR), the redundant modules are often computer-memory pairs. The



# Processes = 3 x (# Nodes + # Arcs)

Figure 1-1: Non-redundant Four Module System and Associated TMR System

computers communicate information to be voted on either by hardware voters [11], or by software voters running on the processors [4] [9]. Software voting has a number of distinct advantages over hardware voting, one of which is the flexibility of the voter. For example, the Software Implemented Fault Tolerant computer (SIFT) [3], has a voter that can handle a 5-way vote, or a 3-way vote. The system can determine which voter to use depending on the number of processors available. The software voter routine can be modified as the system changes in order to improve the system reliability. Other reliability improvement features such as dynamic reconfiguration can be easily implemented in software, and have been shown to improve system reliability [6] [14]. Most of the research on NMR redundancy has made the assumption that the modules are synchronized [1]. Since it is very difficult to force processors to be tightly synchronized, this assumption does not hold for a large class of systems. Some researchers are beginning to realize that asynchronous systems offer distinct advantages in reliability [9] and simplicity. The problem remains though of how to design an asynchronous system that meets the reliability objectives.

A general purpose multiprocessor called Cm* was used to experiment with NMR computer systems [5]. Cm* has an operating system named Medusa that provides primitives for experimentation, and an experimental interactive synthetic workload generator that provides an environment conducive to monitoring Cm* performance. Cm* is a 50 processor (DEC LSI-11s) multiprocessor connected by a hierarchical, distributed switching structure. Each processor is connected to a local memory to form a Computer module called a **Cm**. The processor is connected to the local memory by a switch called an **Slocal**. The Cm's are connected together into **clusters** by a high speed bus. A high speed microprogrammable bus controller, a **Kmap**, controls access to this packet switched bus, as well as providing access to other Kmaps, and their associated clusters. The Kmaps are each connected to other Kmaps by two intercluster busses. All Cms can access all the memory in Cm* through the Slocals and the Kmaps. The memory reference hierarchy consists of local references, intracluster references, and intercluster references.

The Medusa Operating System is a message based, object oriented operating system designed to exploit the architecture of Cm* [10]. It was designed with modularity, robustness, and performance in mind. The functions of the operating system are partitioned into **Task Forces** which are sets of closely cooperating parallel processes. The processes can communicate via messages passed through a communication medium called a **pipe**. Medusa provides for **Conditional** and **Unconditional Sends** and **Receives**. All data in Medusa is stored in system defined objects. Objects can be accessed through private or shared **descriptors**. The Kmap and the Slocal cooperate to convert a descriptor into a physical address. Operating system functions such as message communication, address mapping, interrupt handling, activity multiplexing, and mutual exclusion are performed by the Kmap microcode.

The synthetic workload generator (SWG) [13] is a tool running under Medusa that provides a controllable, interactive experimentation environment for Cm*. The SWG provides a user interface that allows interactive experimentation. The SWG allows the user to vary experimental parameters at runtime, so that experimental data can be collected easily. All experiments for the SWG are represented as a data flow graph. Processes are represented as nodes of the graph. The communication of information is over arcs on the graph. Buffers are used to store messages being passed between nodes. The synthetic workload is made up of repetitions of operations from a **library of actions**. The actions are designed to simulate real operations. Various control structures supported by the SWG facilitate the starting and stopping of experiments. The SWG and the data flow model were used in the experiments described in Sections 4 and 5.

This paper is divided into six sections. Section 1 provided an overview of the field of highly reliable systems, a justification for the work presented, and an overview of the research vehicle and tools used during the research. Section 2 introduces the concepts involved in voting, including synchronization issues, voting frequency, and voted data. Section 3 presents the experimental paradigm. The types of voters used in the experiments are also described. A series of experiments that describe the voting overhead in a TMR software voting system are presented in Section 4. A theoretical framework is developed for a voting overhead model, and experimental results are compared to results predicted by the model. Section 5 presents experiments designed to explore how closely synchronized voting systems must remain. Variation in process execution speed is used to determine how much asynchrony is acceptable in NMR systems. The results of the experiments yield some guidelines for designing asynchronous NMR systems. An analysis of the synchronization data is also presented. Equations are developed that can predict the amount of variation in process execution speed that is acceptable for reliable system operation. A queuing theory model is developed to describe the voter-subtask relationship. The results predicted by the model are compared to the actual experimental results.

2. Voting Concepts

This section is concerned with giving an overview of voting systems and with presenting the issues involved in voting. Triple Modular Redundancy (TMR) was first proposed in 1956 by von Neumann [15]. Since that time, TMR systems have been built and evaluated [2] [7] [14] [16] [17]. Techniques have been used to improve the reliability of TMR systems, and some of these techniques are presented below. In addition, some new concepts that relate particularly to software voting are presented.

The design of redundant systems is intended to improve their reliability by replicating a module N times, and comparing the outputs of the N modules. The comparison should take the N module outputs, and choose the most likely output as the actual output. The comparison has taken many forms over the years, but a simple majority vote is the most popular. A majority ($\lfloor N/2+1 \rfloor$) of the modules must agree on a value for a particular output. Since most computer systems use a binary representation for the data, the voter simply needs to compare the data bit-by-bit.

In an NMR system, if only one voter is used to determine the correct module output, then the failure of the voter becomes a catastrophic event. The voter is called a **single point of failure**. If the voter, however, is also replicated N times then the single point of failure has been removed. The systems considered in this report are all NMR with no single points of failure, generally with $N=3$ (TMR). Systems that are TMR with no single points of failure can **mask** a single permanent, intermittent, or transient error in either the voters or the modules.

If the modules to be replicated are software modules, then each module can execute on its own processor, concurrent with the execution of other modules. The replicated modules that are executing the same task are not necessarily executing at the same time. The replicated modules will have completely separate code and data, so they can be called **space redundant**. In addition, the modules can execute at different times, which is called **time skew redundancy**. If the system uses time skew redundancy, then the system may be able to tolerate multiple failures at one time, since the failures will affect different computational tasks. In a TMR system, three simultaneous failures could be tolerated if the system was both time skew and space redundant, and no more errors occur until the voters correct the three faults.

In order to vote on the outputs of modules, the voters must have some knowledge of when the outputs become valid. Since the modules may have different clocks, the voters must be able to *wait* for modules to prepare outputs, before voting on them. Even if the modules have the same clock (which would be a single point of failure, so should probably be avoided), clock skew and differences in logic delay would introduce the need for the voters to wait for the outputs to all become valid. The wait time could be **implicit**, as in SIFT, such that the vote occurs at a predetermined time (if the module cannot produce the output in time, then the vote proceeds without that output). Conversely, the wait can be **explicit**, as in the Cm* voting experiments presented in Sections 4 and 5, such that the voter waits for a signal from the module indicating the output's validity. In the case of explicit waiting, the voter should not wait indefinitely for the module to signal, since the module may fail in such a way as to never produce the signal. The voter should, in this case, have a **time-out** to prevent indefinite waiting. Two types of time-outs are possible. A module external to the voter could interrupt the voter after a period of time. This requires a clock to determine the time, so is called a **clock driven** time-out. The second possibility is an **event-driven** time out. A number of possible events could trigger a time-out, but in the experiments in Sections 4 and 5 the time-out occurs after the voter receives n messages from one module without receiving any messages from another module.

When the module outputs become valid a voter can determine the majority, and generate its own output called the **voted output**. The point in time when the module outputs all become valid is called a **point of synchronization**, since the system will be synchronized with respect to the module outputs at this point in time. The voter must wait for at least a majority of the outputs before it can decide on the correct voted output, so at least a majority of the modules must reach the synchronization point before the vote. If the voter does not wait for all the modules to generate outputs , but only a majority, then this is called a **point of partial synchronization.**

The amount of work done between votes can be small (a few instructions) or large (thousands of instructions). The trade-off in determining the voting frequency is throughput versus reliability. As the frequency of voting is increased, the overhead due to voting becomes greater. This decreases the throughput, but will increase the reliability. In general, a TMR system can tolerate one error between votes. However, there is a probability that two errors will occur between votes. The assumption will be made that the system can not recover from two such errors. Given an error rate, the system should vote frequently enough so that no two errors arrive between votes. A task to be performed will take longer to execute as the granularity of voting is decreased, due to the overhead introduced by each vote. The probability of two errors occurring between votes will decrease as the granularity decreases, until the voter execution time dominates the total execution time. Any further decrease in granularity will have little effect on the probability of two errors occurring between votes, but the total task execution time will continue to increase. Therefore, the probability of a system failure sometime during the task execution will increase. As the voter takes a larger percentage of the total execution time, the voter becomes the module that is more likely to fail. The system reliability decreases if the granularity is decreased past this point.

There are many issues involved in choosing the amount and kind of data to be voted on by the voter. One of the first decisions made in designing a NMR system is to choose the data to be voted on. Systems can be designed that would vote on the actual data used in a module. The actual data would include processor state that is unimportant to the value of the outputs. For example, if a program is relocatable, then the program counter may be different for each processor. The results produced by the program will, however, be identical. In a system that votes on actual data, the programs being executed must all be placed in the same memory space, and the programs have no flexibility in independently choosing any parameters. A more flexible system might allow modules to act independently, only voting on the parameters that affect the outputs.

Once the data has been passed to the voter, the voter has some options on how to determine the majority. The voter could choose to compare bits, words, or an entire array. The type of data to be compared is called the **data granularity**. The choice of data granularity makes a difference in system reliability. If the data is voted on bit-by-bit, it is guaranteed that a majority will be found. There are only two possible values and one will be the majority. If a majority of the bits are in error then the voted value will be incorrect. If a larger data granularity is chosen, for example an n-bit word, then the voter can reach three decisions. All three can agree on the value, two can agree on the value, or all three can disagree. In this case, the detectability of errors is improved, since the probability of having two incorrect words that agree is less than having two incorrect bits that agree. Word voting is less likely to produce an incorrect answer which may cause catastrophic errors in other modules. The voter can detect when all three disagree, and a recovery routine can decide how to handle the faults. Even though the voter provides no answer, this is preferable to providing the wrong answer. If the data granularity is increased again, then the probability that two incorrect data values agree is decreased. If an entire array is compared to two other arrays, the probability of having two faulty but equal arrays is smaller than the probability of having two faulty but equal words. The array could contain two correctable errors, yet the voter would not correct either because the data granularity is large. The ideal value of the data granularity should be when the probability of having two correctable errors in the data equals the probability of having two incorrect data values agree. A small data granularity allows the voter to correct many errors, and a large granularity reduces the probability of allowing incorrect data to pass the voter. A voter could obtain better detectability and correctability by using a small data granularity to correct errors and a large data granularity to detect errors. This voter would, however have a greater execution time than a simple voter.

Generally, software voters do not vote bit-by-bit, since processors are designed to handle bytes or words better than bits. If the three words passed to the voter are X, Y, and Z, then the combinatorial majority vote is defined as:

$$C = X \cdot Y + X \cdot Z + Y \cdot Z$$

If the values of X, Y, and Z are words, then a bit-wise vote will proceed in parallel for all n bits in the word. The generation of the voted data value with this method takes just three bit-wise **AND** operations and two bit-wise **OR** operations. The comparison voter that is popular in many software voting systems requires at least three comparisons, and two branches. The **combinatorial majority voter** has straight in-line code that could be pipelined on a special purpose machine to improve performance, where the comparison voter can not be pipelined. The combinatorial majority voter therefore requires less execution time than the classical comparison voter, and increases the probability of correcting independent errors.

In addition to choosing the data granularity, other parameters of the data must be chosen. It may be desirable to vote on some abstract data structures, to determine if the data they contain is equal. Some interesting problems arise, due to the nature of some data structures. For example, a linked list data structure may be passed to a voter by three modules. The voter should vote on the data in the linked list, but should not vote on pointers to data items. The lists should have the same structure, and the same data, but not necessarily the same pointers. This procedure requires an intelligent voter, with knowledge of linked lists, and with knowledge of the storage format. Other interesting data structures, such as queues or stacks could be used as inputs to the voters. Abstract data structures are commonly used in high level programming languages, so the voters should be able to handle them. An NMR system should attempt to accommodate the programmer, not the other way around. Although no systems provide abstract voting yet, as more applications are written for NMR systems, the programmers are going to discover the advantages of having voters that can handle abstract data.

3. Experimental Paradigm

The two types experiments performed use a similar paradigm. The paradigm can be viewed at the highest level as the execution of a single task. The task to be performed is broken into equal subtasks. Each subtask is executed in order, with data being passed from one subtask to the next. It is assumed that each subtask has the exact same execution speed, and that only one word of data is passed from one subtask to the next. Since the subtasks all have the same execution speed, the task can be simulated by a loop that executes n times with a synthetic workload that takes subtask_t time inside the loop. Figure 3-1 shows the partitioning. Each subtask

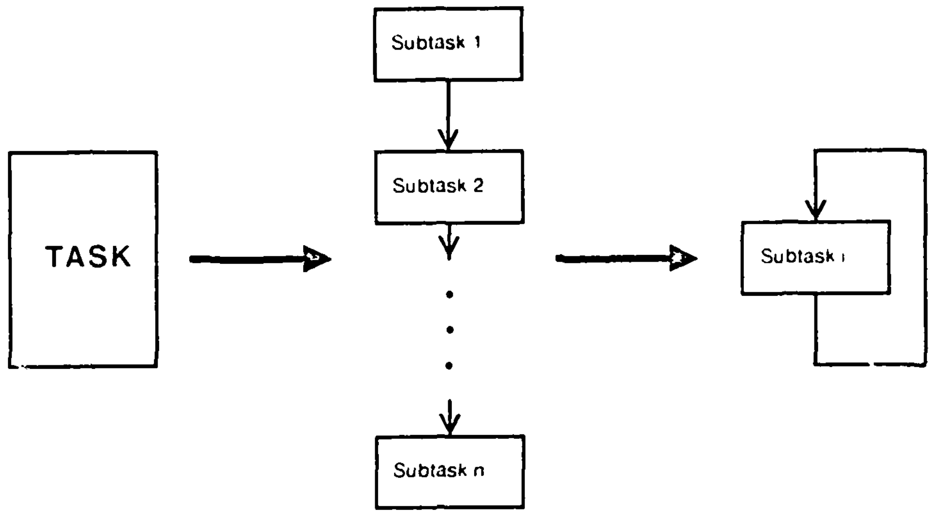


Figure 3-1: Experiment Task Partitioning

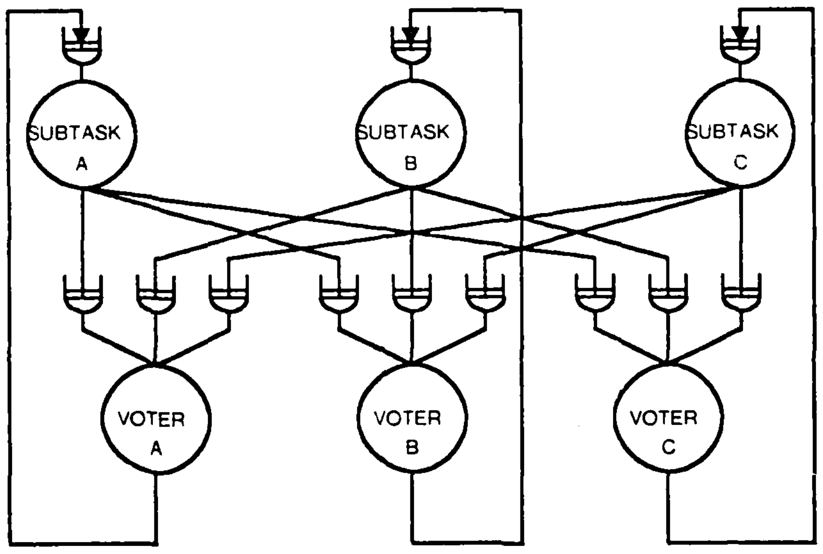


Figure 3-2: TMR Experimental Structure

is triplicated, and a vote occurs on the data passed between subtasks, yielding the structure in Figure 3-2.

The triplicated subtasks all perform the same function. They will calculate the i^{th} data value, send a copy of the data to each voter, and receive the voted value of the data from the associated voter. The new data value is then used in calculating the $(i+1)^{st}$ data value.[1] The time each subtask takes to calculate the i^{th} data value is an experimental variable. All of the triplicated subtasks will have a variable execution time. This time is set by the **granularity** of the subtask, which is defined as the number of operations executed between votes not including the overhead due to voting. An operation is four LSI-11 instructions. The granularity of each subtask can be set before an experiment.

The voter subtask is also triplicated, as shown in Figure 3-2. Each subtask sends each voter two data words. The first data word is a sequence number to associate data with an iteration. The second word is the data to be compared by the voter. When a voter has received data from a majority of the subtasks (two), it checks to see if the data values agree. If so, then a majority vote has been achieved, and the data value is sent to the subtask associated with this voter. If they do not agree, then the voter waits for the data value from the third subtask to determine the correct value, which is sent to the associated subtask. Each voter and subtask is assigned its own processor, so each voter proceeds with the voting in parallel with the subtask execution.

Three types of voters are used in the experiments. The first voter, called the simple voter, is a synchronizing voter. It requires the subtasks to reach a full point of synchronization after each subtask iteration. It has no internal storage of data from one iteration to the next. The second voter, called the internal queue voter, has an internal queue that allows it to handle data from different iterations. The subtasks are not required to fully synchronize after each iteration. This voter has been optimized for high execution speed in the average case and therefore has the shortest execution time. The third voter, called the sequence number voter, uses the sequence numbers that are sent by the subtasks, so that the voter can order data based on the subtask iteration. This voter has the longest execution time. All three voters were designed to allow easy expansion to N-way voting. The algorithms for the voters are presented in Appendix I.

As long as the subtasks have similar execution speeds, the voter should receive the i^{th} iteration from each subtask at approximately the same time. The sequence number voter and the internal queue voter do not require a full point of synchronization, so if one subtask is slower than the other two then the voter may receive the $(i+1)^{st}$ data value from a fast subtask before the slow subtask sends the i^{th} data value. Since the voter now has data from two different iterations, it must be able to distinguish which data is associated with which iteration, and from which subtask. A voter queue is used to maintain this database. Each row in the queue contains information about:

[1] One can imagine wanting to pass more than one data value from one subtask to the next. This can be done with a more complicated voter. The entire state of a processor (or selected parts) could be passed as data, allowing a faulty processor to recover from a transient by accepting the voted state as its new state. Adding this capability to the experiments would complicate them without yielding additional information about the voting.

1. which iteration this row represents.

2. whether data has arrived from each source subtask.

3. what the data value is from a source subtask (if it has arrived).

The column the data is stored in implicitly identifies the associated destination subtask.

The sequence number voter then searchs for an iteration number in the voter queue to find the row where the data for this subtask belongs. If the iteration number is not found in the queue, a row for this iteration is placed in the queue and the data is placed in the row. When all of the data values for a particular row have arrived, the voter reports any errors found while voting and then removes the row from the queue.

The voter queue has a finite maximum length. If one subtask has not sent any data to the voter in the same period in which the other two subtasks have sent many data messages, the voter queue could conceivably become full. The voter handles a full queue by removing the oldest row (associated with an iteration for which all the data has not arrived) from the queue and adding a row associated with the new iteration number. Errors are reported on the row removed from the queue. The maximum length of the queue can be large, so that the queue will never become full in experiments.

4. Voter Overhead Experiments

In any N-modular redundancy (NMR) system, the amount of useful work done will be less than the corresponding non-replicated system. The voting that is done in a NMR system will introduce some overhead that will reduce the system throughput. The overhead will be made up of many different components, including the communication time between modules and the time required by the voters to receive messages and find the majority. In this section, voting overhead is discussed and a model is developed to describe voting overhead.

In order to develop a model for voting overhead one must determine a method for representing overhead, and must determine what parameters affect the overhead. One possible representation for overhead is in additional operations executed per unit time (operations/second) due to redundancy. The actual throughput is the number of subtask operations performed per unit time. As the actual throughput goes down, the overhead goes up. Mathematically:

$$\textit{Overhead} = \textit{Non redundant Throughput} - \textit{Actual Throughput} \qquad (1)$$

In this section, the actual throughput is determined, and the overhead can be calculated from the above equation. The non-redundant throughput is a constant for a given system.

Each subtask is executing an instruction sequence iteratively. Since each iteration is identical, the total overhead is the number of iterations times the overhead for one iteration. A subtask performs work for each iteration and the amount of work is called the granularity, G. Since the total amount of work to be performed is a constant, W, then the number of subtask iterations, I, is:

$$I = W/G \text{ or } W = I \cdot G \tag{2}$$

In other words, if the total work is 100 units, and 5 units are performed per iteration, then 20 iterations must be performed.

As an experiment is performed, the total execution time is measured. The execution time, t_T, is the time from when a subtask begins the first iteration until the subtask finishes the last iteration. The Throughput, T, therefore is:

$$T = W/t_T \tag{3}$$

The total time, t_T, can be expressed as:

$$t_T = t_{i-ave} \cdot \mathit{Total\ number\ of\ instructions} \tag{4}$$

where t_{i-ave} is the average instruction execution time, and

$$\mathit{Total\ number\ of\ instructions} = I \cdot (a \cdot G + k) \tag{5}$$

where a is the number of instructions executed by a subtask when $G=1$ and k is the total overhead per iteration, including voting. Therefore from Equations 4 and 5

$$t_T = t_{i-ave} \cdot I \cdot (a \cdot G + k) \tag{6}$$

From Equations 2 and 3,

$$\begin{aligned} T &= \frac{W}{t_{i-ave} \cdot I \cdot (a \cdot G + k)} \\ &= \frac{1}{t_{i-ave} \cdot (a + k/G)} \end{aligned} \tag{7}$$

The throughput, then, is inversely proportional to the average instruction execution time, the number of instructions per subtask iteration, and the number of overhead instructions over the Granularity (k/G). The values of k, t_{i-ave}, and a are experimental constants, so we can plot the throughput versus the granularity. For typical values of k, t_{i-ave}, and a ($k=800$, $t_{i-ave}=6.5\mu s$, $a=4$), the curve is shown in Figure 4-1.

The previous overhead model is both general and accurate. Although selection of the value of k (the subtask overhead per iteration) is difficult, a careful approximation to k can be found.

Cm* was used as the experimental vehicle. The Voter and Subtask software routines were triplicated and each placed on their own processor. The number of iterations, I, and the granularity, G, were varied during

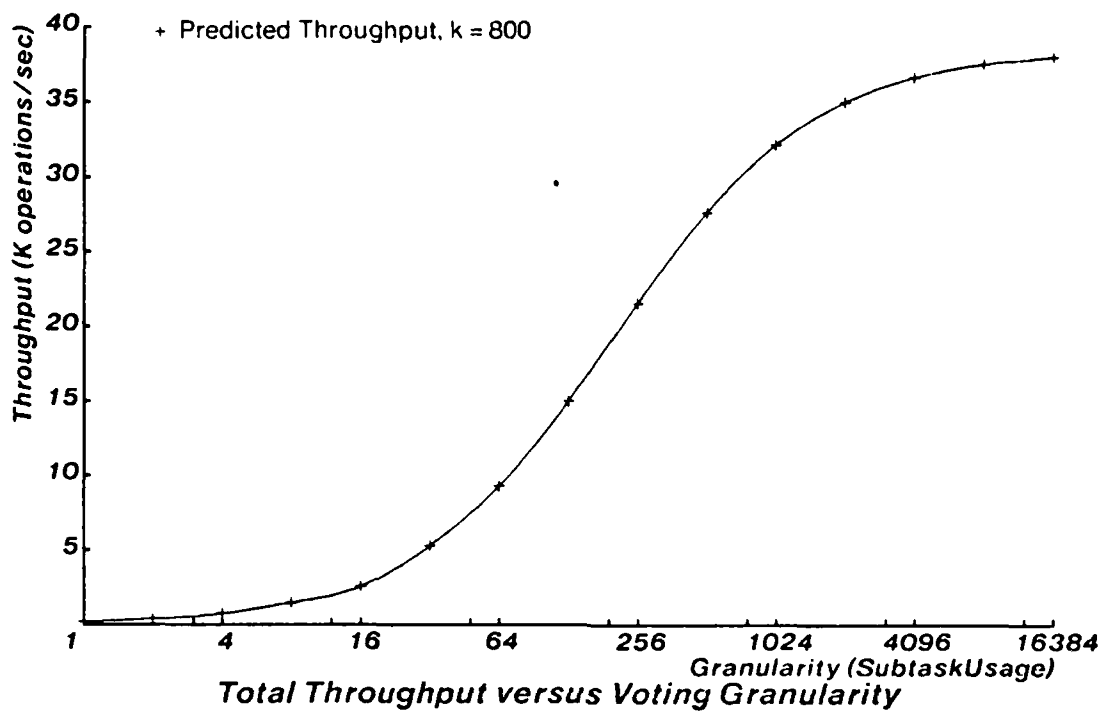


Figure 4-1: Predicted Voting Overhead

the experiments. The execution time, t_T, was recorded for each set of values of I and G. The total work done, W, was kept constant by chosing a value of I and calculating the value of G. The value of W was chosen to be 16,384 operations. The throughput was calculated for each execution time. All three types of voters described previously were used in this experiment. From the overhead model is can be seen that changing the type of voter should only affect the value of k in Equation 7. The throughput versus the granularity is plotted for various voter changes in Figure 4-2. Even when the voter is changed significantly, the change in throughput seems to be small.

The model is extremely accurate in predicting the overhead in a system. One problem with the model, hinted at earlier, is the difficulty in finding values for the constant k. The value should be predictable by adding the instruction execution times in the Subtask and the Voter, but some of the instructions used do not have predictable execution times due to factors like system load. In addition, some of the voting and subtask execution are performed in parallel, so instruction counts would give an upper bound on k, but not an accurate value. The amount of parallelism is difficult to quantify without seriously perturbing the experiment. Therefore, the value of k used in Figure 4-1 was estimated using experimental results. The

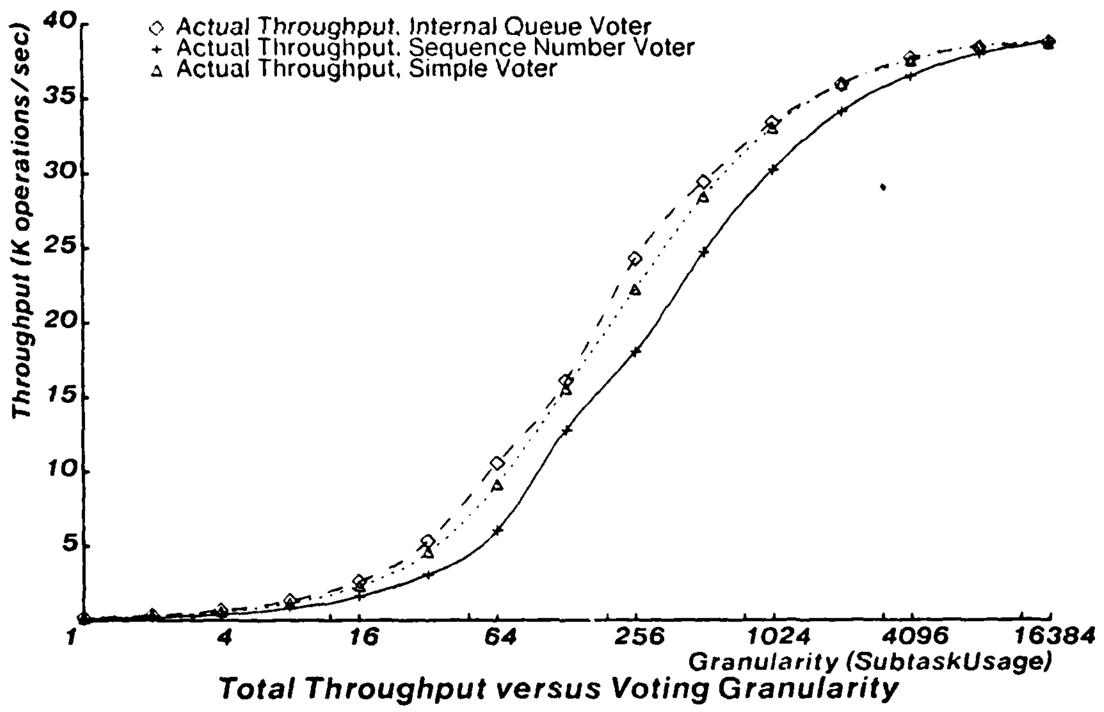


Figure 4-2: Actual Voting Overhead for Various Voters

comparison of the predicted and actual curves, however, loses credibility since the values of k for the predicted curves must be experimentally determined.

The value of k can be given an upper bound for the non-error case. The upper bound will change as the voter changes, but for any given experiment the upper bound can be determined. For the optimized voter and subtask experiment, this upper bound has been found by adding the instruction execution times for the subtask overhead and the voter time. The actual value of k will be less than this time because the voter will be executing simultaneously with the subtask. An upper bound on k is approximated by:

$$k_{max} = k_{s-max} + k_{v-max}$$

where k_{s-max} is the maximum subtask contribution to k and k_{v-max} is the maximum voter contribution to k. By analyzing the programs written for the experiments, it is found that:

$$k_{s-max} = 68\ \textit{instructions} + 3\ \textit{Sends} + 1\ \textit{Receive}$$

$$k_{v-max} = 237\ \textit{instructions} + 3\ \textit{Conditional Receives} + 1\ \textit{Send}$$

The execution times for sends and receives on Cm* Medusa are given in [12]. The average execution time for

LSI-11 instructions in the voter and the subtask was determined to be 6.5μs. Using this information, k_{max} is determined.

$$k_{s-max} \approx 333 \text{ LSI-11 instructions}$$

$$k_{v-max} \approx 471 \text{ LSI-11 instructions}$$

$$k \leq 333 + 471 = 804 \text{ LSI-11 instructions} \qquad (8)$$

Similarly, the lower bound can be approximated by:

$$k_{s-min} = 68 \textit{ instructions} + 3 \textit{ Sends} + 1 \textit{ Receive}$$

$$k_{v-min} = 127 \textit{ instructions} + 2 \textit{ Conditional Receives} + 1 \textit{ Send}$$

$$k_{min} = max(k_{v-min}, k_{s-min})$$

$$k \geq 333 \text{ LSI-11 instructions} \qquad (9)$$

Equation 9 assumes maximum simultaneous execution of the subtask and the voter. The experiments with the optimized voter yielded values of k between 350 and 712. These experimental results fall between the minimum and maximum theoretical values calculated above. The bounds should be recalculated if the voter or subtask is changed. Figure 4-3 compares the minimum and maximum predicted curves, and an experimental curve (for the optimized voter). One result that the model does not take into account is that the value of k changes as the Granularity changes. During the optimized voter experiment, the value of k varied by over 350 instructions. This is due to the change in load on the Kmap processors as the Granularity changes. The model assumes that the value of k stays constant throughout the experiment. In spite of these deficiencies, the overhead model does give accurate predictions of expected voting overhead.

5. Voter Queue Length Experiments

In an asynchronous NMR computer system, the processors will have their own clocks and will make little or no effort to synchronize the clocks with each other. The random variation in clock speed and the difference in process execution patterns will cause differences in the arrival times of the data to be voted on by the voters. The voters should be able to receive data asynchronously so that they can vote on the data when a majority of the processes have sent it. The voters must be able to store message values so that one processor can be calculating the 10th step in a procedure while another processor can be working on the 12th step. Eventually both processors should finish the procedure but as long as no data dependencies exist, one processor should not be forced to wait for another to finish a calculation. Even when data dependencies do exist, when a majority of the processors agree on the value of a step, there is no reason to wait for the rest of the processors to finish before continuing with the next step. In fact, waiting can reduce reliability if a

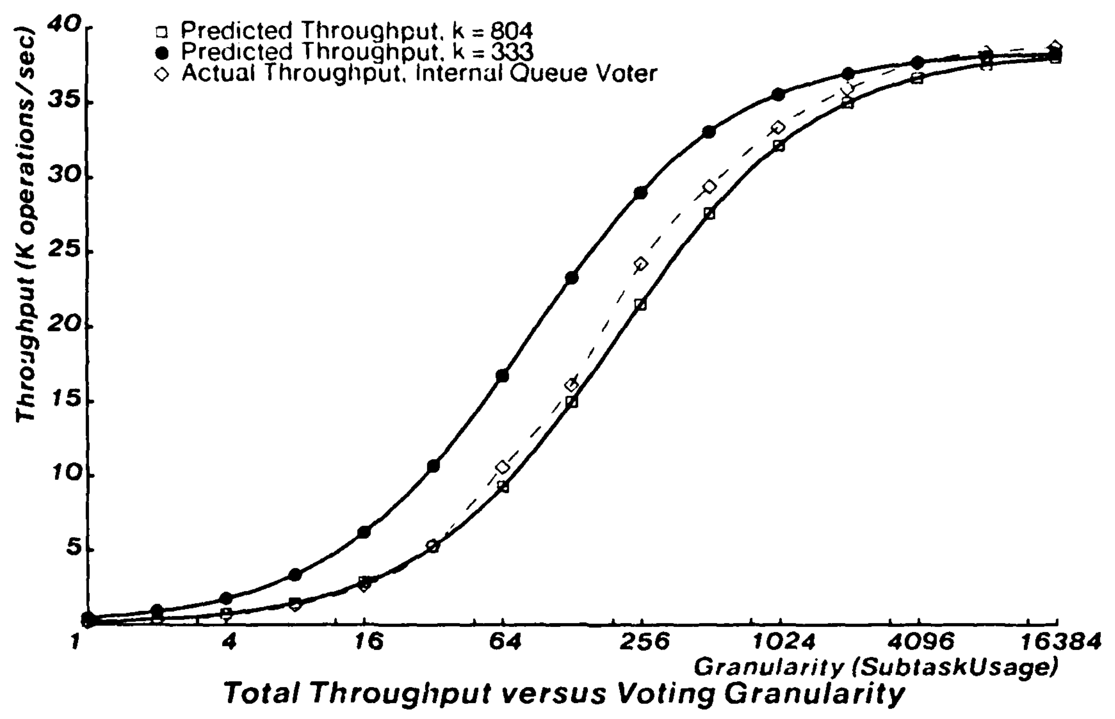


Figure 4-3: Comparison of Actual and Predicted Voting Overhead

processor is faulty since it may never respond to the voter. There should, however, be a limit to the amount a processor should be allowed to fall behind before it is considered faulty. The random variation may cause problems if one processor becomes hopelessly behind due to the variation. Experiments have been performed to discover the nature of how variations in process execution speed affect the amount a process falls behind the others. The effects of variation in process execution speed, as well as variation of the number of instructions executed between votes have been examined.

Three experiments have been performed. Each is designed to explore a different area of the synchronization problem. Experiment one has a single process execute more instructions for every step in the experiment. This process is continuously slower. This experiment shows that the voter overhead increases as the slow process falls behind. Experiment two has one process slower for a period, followed by being faster for a period. Experiment three has one process slower for a period, followed by a period of normal speed. This experiment is realistic for many systems, since processes are likely to fall behind in a system but are not likely to speed up.

5.1. Experiment One

The first experiment performed was designed to measure the ability of the voter to synchronize the subtasks when one subtask is continuously slower than the other subtasks. The frequency of voting (or granularity of the subtasks) was varied, and the execution speed of one subtask was varied. The queue lengths of the voters were recorded as a measure of how far the slow subtask fell behind the two faster subtasks. The slower subtask performed 10% to 50% more operations in calculating the next value. The slower subtask represents a process that requires more execution time due to an instruction retry, or due to an interrupt that it must handle. In these situations, one subtask will be temporarily slower; but as these experiments show, it would be ill-advised to design a system where one subtask was continuously slower (this experiment shows design constraints for systems that have one continuously slower subtask). Each voter recorded the length of the voter queue every time a new iteration was received. The queue length information was sent as a message to a process that stored the data in a file. The recording of the queue length added some overhead to the voter, but each voter paid the same cost.

The queue length was plotted versus the iteration number for two different granularities and various subtask degradation as shown in Figures 5-1 and 5-2. For granularity equal to 1024 operations, one subtask can be up to 10% slower and the queue length stays at one. This implies that the voter overhead is great enough so that the differences in speed are masked. For larger differences in speed, the queue length grows to a value and then levels off. The queue length is bounded due to an increase in voter execution time as the queue length increases. The voter must search for the iteration number in the queue and the search proceeds linearly. The subtask that is slower will not pay this overhead cost since it has *n-1* messages waiting for processing, where *n* is the queue length.

As the granularity increases, the queue length grows more rapidly. With granularity equal to 1024 (Figure 5-1), the 10% to 40% additional operations curves appear to be bounded but the 50% additional operations curve is not bounded. The curves for granularity equal to 16,384 (Figure 5-2) do not appear to have a bounded queue length. This is due to the fact that the voter overhead takes a smaller percentage of the total execution time for the larger granularity cases. The voter overhead is a fixed value for a specific queue length. When the slower subtask takes approximately the same amount of time as the voter, then the voter overhead is significant in comparison to the subtask execution time. While the normal subtasks are waiting for the voter to generate a voted data value, the slower subtask can be calculating a data value for one of the old messages (when the queue length is greater than one, the slower subtask will have data values to calculate for all the messages in the queue).

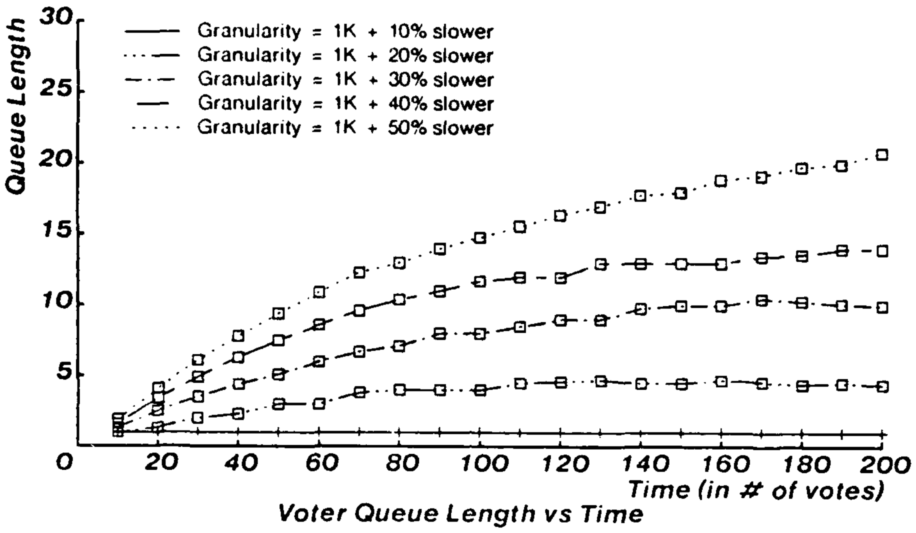


Figure 5-1: Granularity equal to 1024, one subtask always slower

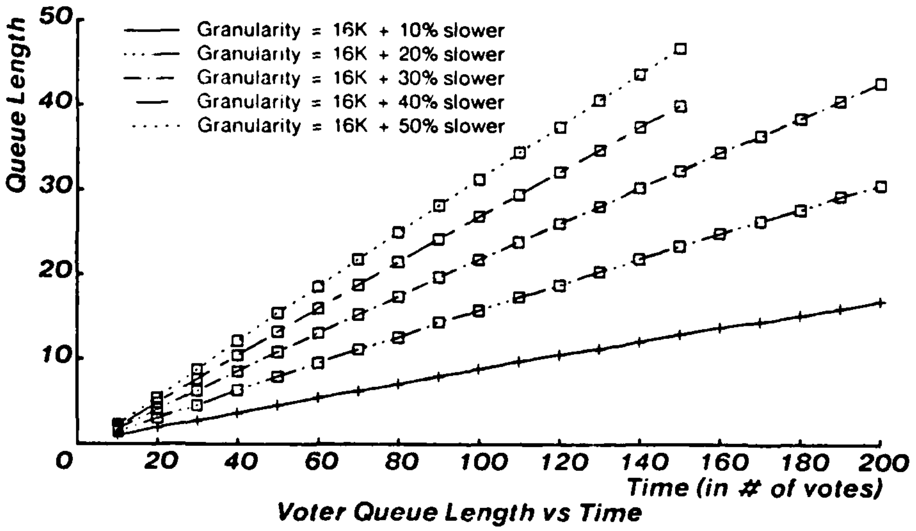


Figure 5-2: Granularity equal to 16,384, one subtask always slower

5.2. Experiment Two

The second experiment is a variation on the first experiment and was designed to explore the synchronizing nature of voters more fully. In this experiment, one subtask is slower than the other two subtasks by a percentage for a period of time, then the same subtask is faster than the other subtasks for the same period. The period was chosen to be 20 iterations. For example, subtask A will perform 10% more operations in calculating the first 20 data values, followed by performing 10% fewer operations for the next 20 iterations. Subtask A will therefore spend 10% more time executing the first 20 iterations than the second 20 iterations.

While the subtask is operating slower, the queue length should behave exactly the same as in experiment one. Once the subtask is faster than the others, this subtask should quickly catch up resulting in a decline in the queue length. The rate of decline in queue length should be greater than the rate of increase, since when the queue has length greater than one the subtask being varied does not have to wait for the voter to finish before beginning the next data value calculation.

The first plot of queue length versus iteration number with granularity equal to 1024 (Figure 5-3) shows the expected result. The queue length increases when subtask A is slower and the rate of increase is the same as that from experiment one. As soon as subtask A begins executing fewer operations per iteration, the queue length declines rapidly, reaching queue length equal to one. If the granularity is increased to 16,384 (Figure 5-4) then the queue length is not restored to one, and there is a net increase in the queue length over time. The queue length increases because subtask A will be spending more time executing the long calculations.

5.3. Experiment Three

The third experiment is similar to experiment two, except it represents a more realistic class of synchronization problems. A subtask that is performing a calculation may experience a temporary slowdown, followed by a period of normal behavior such as a subtask which has to perform a recovery routine because of a bus error or has to perform a one time operating system task. Is the processor running the subtask doomed to stay behind, or will it eventually catch up even though it always takes as long to calculate a new data value as the others? As soon as a subtask falls behind, it no longer pays the overhead cost since it has messages queued up waiting for processing. This fact would imply that a subtask can catch up, and the rate at which it catches up is the incremental voter overhead cost per iteration.

The experiment can be described as follows: one subtask will do additional operations (10% to 50%) for 20 iterations followed by a period of normal behavior (performing the same number of operations as the other subtasks). The results of the experiment are shown in Figures 5-5 and 5-6. It can be seen that during the periods of normal operation for all three subtasks, the queue length declines, and given a long enough period

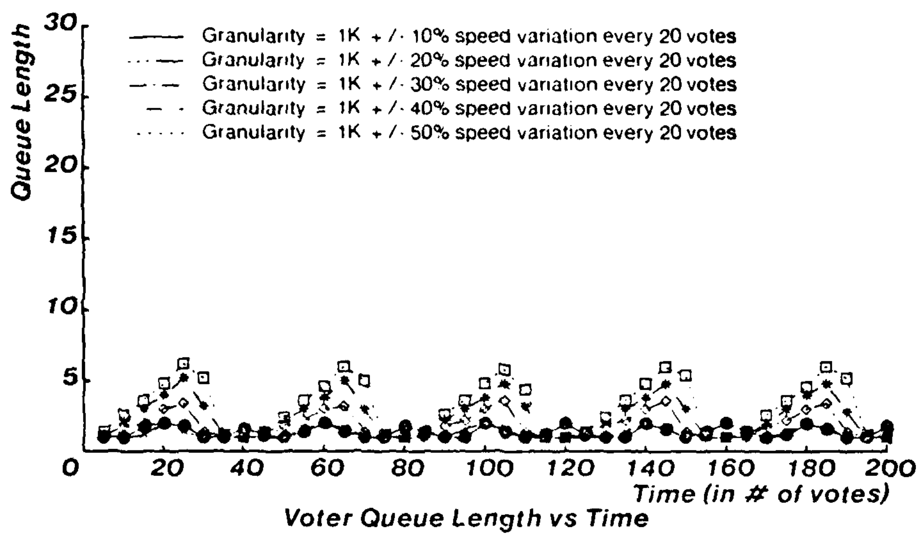


Figure 5-3: Granularity equal to 1024, one subtask slower half the time, faster half the time

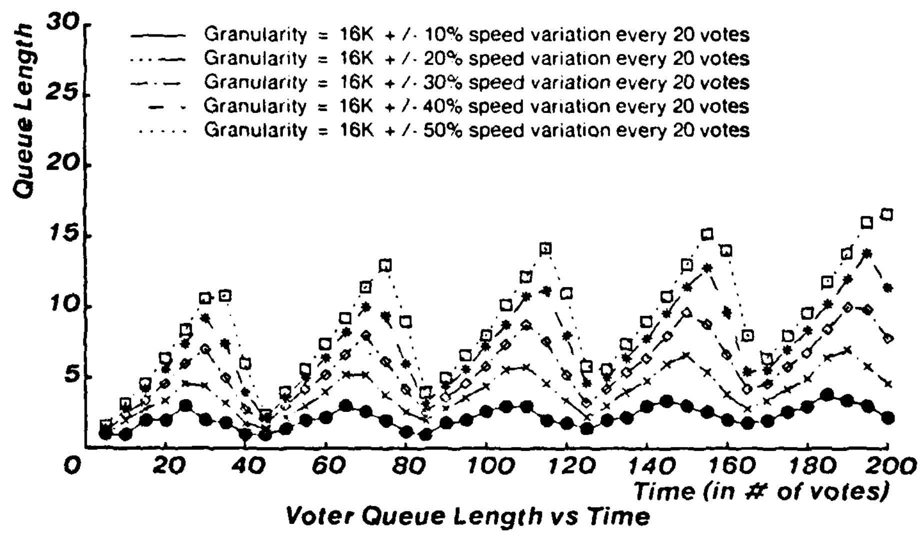


Figure 5-4: Granularity equal to 16,384, one subtask slower half the time, faster half the time

of normal behavior would reach one. The rate of decline of queue length during normal subtask behavior indicates the effect of voter overhead on the subtasks.

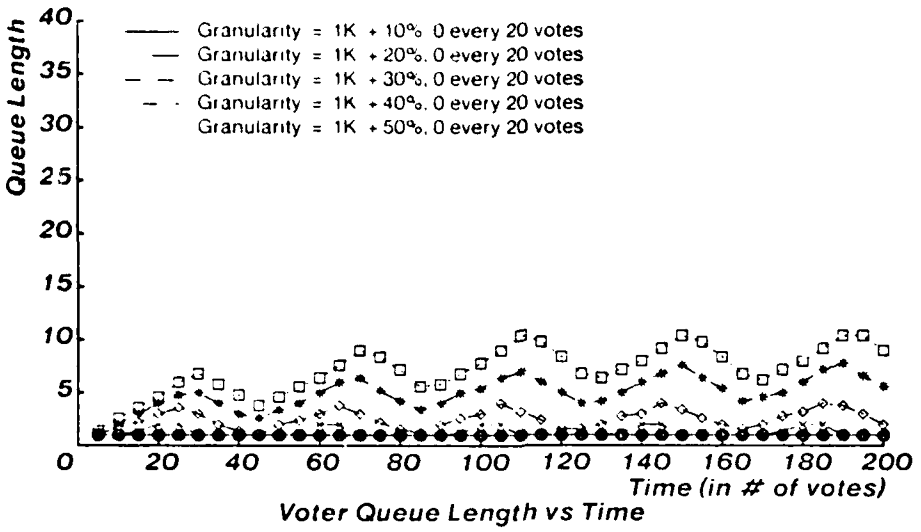


Figure 5-5: Granularity equal to 1024, one subtask slower half the time, same half the time

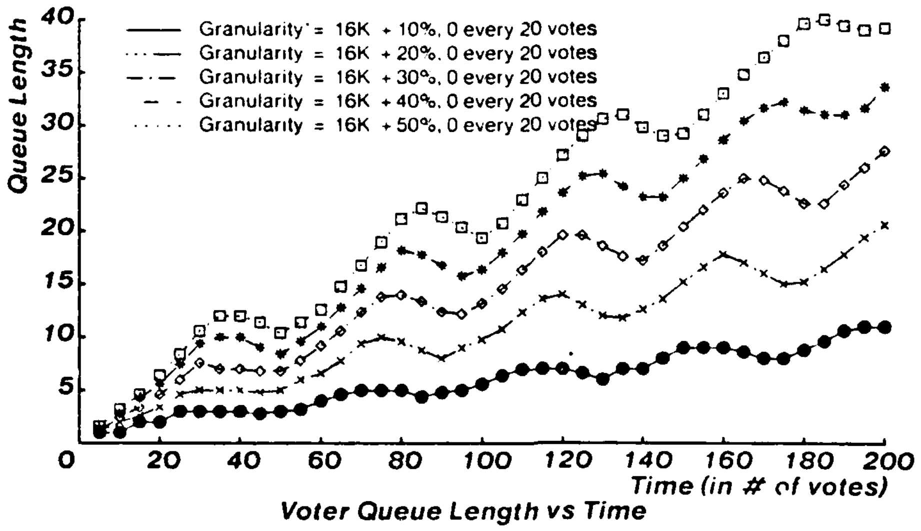


Figure 5-6: Granularity equal to 16,384, one subtask slower half the time, same half the time

5.4. Experimental Conclusions

The three experiments performed give a clear picture of a synchronization model for the equal subtasks paradigm. There appear to be two factors involved in the model. The factors are:

1. There is a minimum voter overhead that is due to the time required by the voter to receive a message, handle the data, and vote on the data. The subtasks that have a queue length of one must pay this overhead cost every iteration of the experiment.

2. The overhead cost increases as the voter queue length increases due to an increase in the data handling cost. This factor would indicate that for a long enough queue, the voter could mask any difference in process speed. For practical queue lengths, though, the increase in voter overhead masks only some of the subtask speed variation.

The synchronization experiments can give some design principles for TMR asynchronous voting systems. These principles can be applied to optimize the voter queue length, to choose a subtask granularity, and to determine the amount of process speed variation allowed in a design. Proper application of the principles will lead to a design that will have a bounded queue length for all possible variations in process execution rate. The principles can be summarized as follows:

1. Smaller granularity subtasks have a higher probability of having a bounded queue length.

2. As subtask granularity increases, the random variation in process speed becomes increasingly important in ensuring a bounded queue length.

3. Greater voter overhead allows a greater variation in process execution rate. This yields an interesting trade-off in voter design, since a faster voter process will increase system throughput but will decrease the amount of variation permitted in process execution rate.

These results can be generalized for synchronous voting, as well as asynchronous voting. If the maximum voter length is fixed at one, then the system is synchronous like SIFT [2] [3] [4] and C.vmp [8] [11]. Both of these NMR systems use a synchronous voter with queue length of one. C.vmp has a hardware voter with a built in wait feature. The length of the wait corresponds to the voter overhead in these experiments. SIFT uses fixed scheduling, so a vote proceeds when the next time slot begins. The voter overhead corresponds to the design margin in the fixed schedule (the time between the end of the process execution, and the end of the time slot).

5.5. Synchronization Modeling

This section will present a model of the voter queue length based on granularity, percent difference in subtask execution speed, and time. The model is compared to the actual experimental results, but first the relationship between the model and the FMR experiment should be explained.

5.5.1. Queue Length Models

The FMR system explained in the previous section has queues that contain the messages being passed between the subtasks and the voters. Each voter has three queues in which to receive messages, and each subtask has one queue in which to receive messages. The subtask message queue can be viewed in the light of general queuing theory. The queue will have a birth rate, λ, and a death rate, μ. Basic queuing theory assumes that both λ and μ are constant. Also, the birth rate must be less than the death rate so that the queue length will be bounded. The servers of the queue have a utilization of λ/μ. The utilization will be less than one. There are two problems with using a simple queuing model for voter synchronization. They are that the birth rate, λ, is not constant and that the birth rate is not less than the death rate for most of the experiments performed (the queue length grows, therefore λ is greater than μ) . In spite of these problems, a queuing model can be developed.

Before a queuing model is presented, some background analysis of the previous section's data will be done. Experiment one, in which one subtask was continually slower, will be used in developing the model of queue behavior. Each experimental curve in the previous section begins to peak as time proceeds. The queue length grows less rapidly as the queue length increases. The queue length appears to approach some bound that is dependent on the granularity and the difference in execution speed. Some curves have observable bounds. The information from all the experiment one curves presented could be summarized if this bound information could be collected. If the queue had a maximum possible value, then each curve either remains below the maximum or rises above the maximum. If a curve has a maximum value greater than the maximum queue length, then the queue will overflow during the experiment, and is **unbounded** by this queue length. Otherwise, the curve is **bounded** by the queue length. For three different maximum values of the queue length, the bounded regions and unbounded regions are shown in Figure 5-7. In designing a system, the maximum queue length can be chosen, and this will determine the acceptable granularities and subtask execution speed differences to prevent the queue from overflowing. The curves that determine the regions appear to be linear on the log versus log scale. That implies that:

$$\log_2 Granularity + \log_2 PercentDifference = constant$$

therefore,

$$Granularity \times PercentDifference = constant = VoterOverhead$$

This result indicates that for a given queue length, the granularity of the subtasks is inversely proportional to

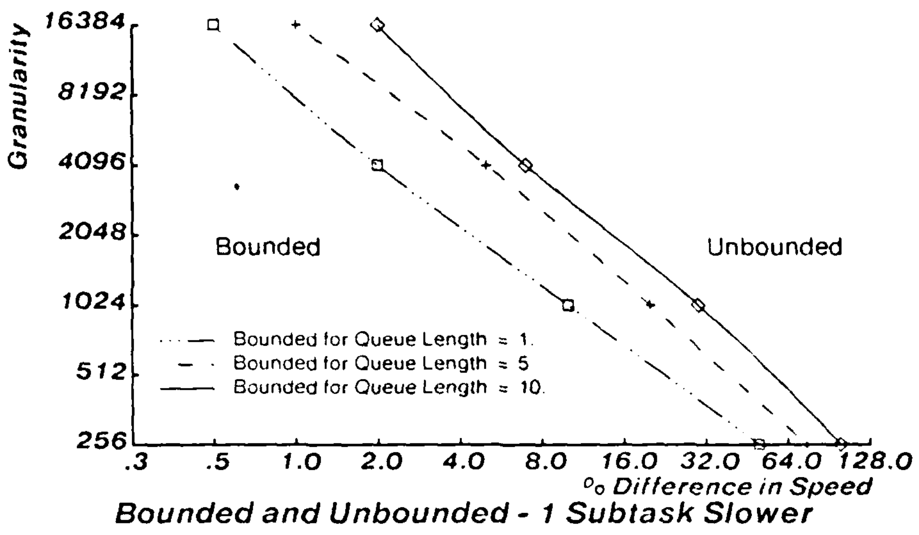


Figure 5-7: Summary of Experiment One Data

the percent difference in processor speed. The constant is a number of operations which is dependent on the voter overhead. A first approximation would equate this number of operations to the voter overhead for one iteration. The voter overhead is constant along a boundary separating the bounded and unbounded regions. A subtask can be constantly slower by a number of operations (the voter overhead) and still only fall some constant number of iterations behind the other subtasks. Next, the value of the bound can be determined for any granularity and percent difference.

From the experimental data, the value of the voter overhead per iteration (the number of operations slower one subtask may be and not fall further behind) can be plotted against the bound on the queue (the maximum queue length). Figure 5-8 shows the data. A linear least squares fit was determined for the data. This equation can predict the maximum queue length for a given granularity and percentage difference in subtask speed. The equation is:

$$MaximumQueueLength(M) = 0.0457 \cdot VoterOverhead - 4.0 \qquad (10)$$

or

$$VoterOverhead = 21.0 \cdot M + 92.2 \qquad (11)$$

Equation 10 is fairly accurate in predicting the bound on the queue, but some of the variation in the data remains unexplained. Equation 11 can predict how much variation in subtask execution speed is allowed given a maximum queue length. Note that even when the maximum queue length is zero (a totally synchronous voting system), some variation in subtask execution speed is allowed. *In fact, this model*

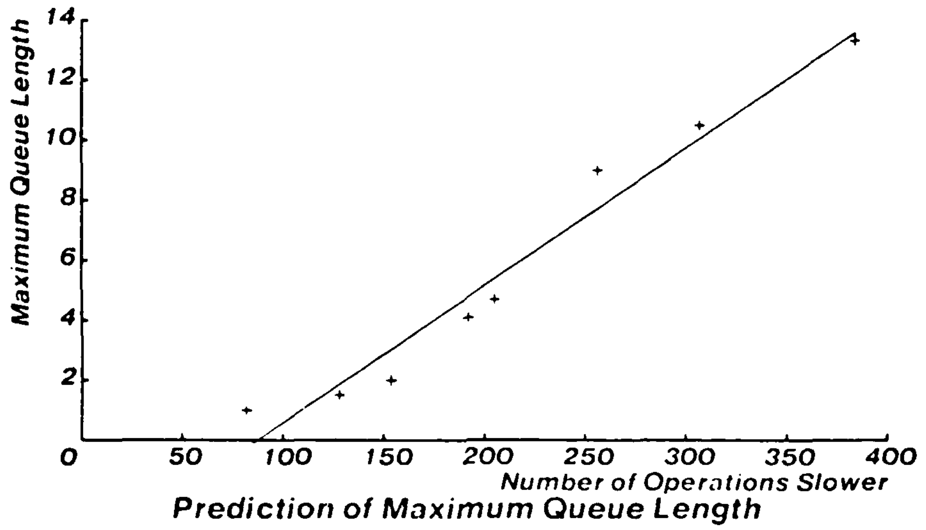


Figure 5-8: Maximum Queue Length

indicates that one subtask can be constantly 92 operations slower and never fall behind. This is the minimum voter overhead, the time the voter takes to process two inputs. This overhead is one component of the value of k presented in Section 4.

5.5.2. Queuing Theory Model

A subtask that has a slower execution rate than the two other subtasks will fall behind in executing each subtask iteration. For every iteration the slow subtask is behind, the subtask queue will contain a message. The queue length will grow as long as the subtask execution rate is greater than the voter execution rate. In the previous section it was shown that the voter execution time is dependent on the length of the queue. In fact, as the queue length grows, the voter takes longer to execute. This will result in a decreasing growth rate for the subtask message queue. Now a model can be formalized.

l = the queue length
$\lambda(L)$ = the birth rate, a function of the queue length
μ = the queue death rate

if $\frac{\lambda(L)}{\mu} > 1$ then the queue grows

when $\frac{\lambda(L)}{\mu} = 1$ then the queue length is in steady state

$\lambda(L) - \mu = g(L)$ = the growth rate of the queue length

$$g(L) = x(M - L)$$

$M =$ maximum queue length in steady state
and
$x =$ percentage decrease in growth rate for each unit increase in L

The maximum queue length equation was derived in the previous subsection. Using this, the value of x can be determined. At the start of each experiment, the length, L, will be zero. So,

$$g(L) = xM \quad \text{when} \quad L = 0$$

This model indicates that the initial growth rate is only dependent on the maximum queue length, and a constant percentage. The growth rate is simply the slope of the curve. Since the growth rate can be experimentally determined, the value of x can be found. The growth rate when $L = 0$ was determined for a number of the experimental curves. From this information, the value of x was determined to be:

$$x \approx \frac{16}{Granularity}$$

This result has no known significance, but is accurate over all values of granularity and percent difference in execution speed considered in the experiments.

Using the above results, the growth rate, which is simply the change in queue length over time, can be written as:

$$g(L) = \frac{dL}{dt} = x(M - L) = \frac{16}{Gran}(M - L)$$

The value of L in the above equation is a function of time, so:

$$\frac{dL}{dt} = \frac{-16}{Gran} L(t) + \frac{16M}{Gran} \qquad L(0) = 0$$

The solution to this differential equation is:

$$L(t) = M\left(1 - \exp\left(\frac{-16 \cdot t}{Gran}\right)\right)$$

Since M is experimentally known, then the queue length can be plotted against time, for various granularities, and percent differences in subtask execution speed.

5.6. Comparison of Model and Experiment

Five experimentally determined curves are compared to five predicted curves in Figure 5-9. The predicted results are very similar to the experimental results. The model seems to be good at predicting the queue length. This model does not, however, take communication costs into account. When the granularity is small, the Cm* interprocess communication costs become significant causing each subtask iteration to have a greater execution time. Therefore the model is not accurate for small granularities. Another problem with the model is that it can sometimes predict a maximum queue length too large, and at other times can predict a maximum

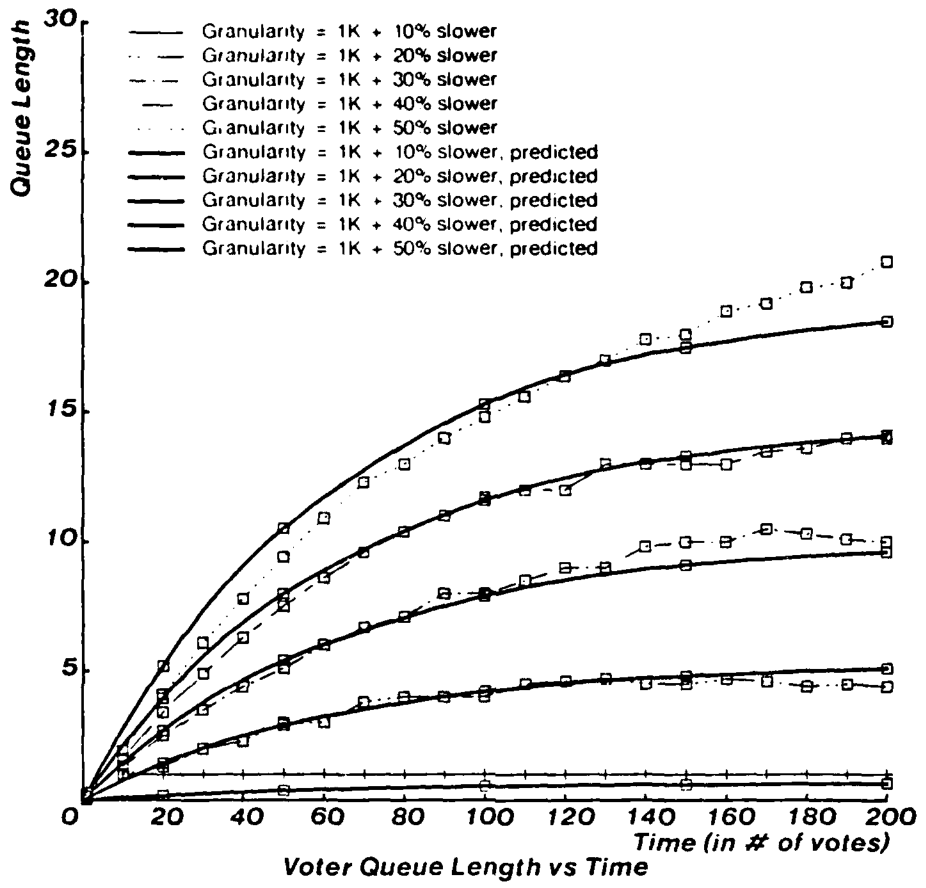


Figure 5-9: Predicted Queue Length and Actual Queue Length

queue length too small. The predictions are not consistently too high or too low. The model has several derived parameters. The value of x was found experimentally, and the equation found for calculating M is based on a least-squares fit in which some points are outlying.

The queuing model of the voter synchronization experiments can explain a large portion of the variation in the experimental results. Some of the model parameters are difficult to determine, but they can be approximated. The comparison of the predicted and actual results shows that the model has the proper form in order to explain the experimental results. By changing the model parameters slightly to account for Cm* perturbations, the model can explain most of the experimental results.

6. Conclusion

This paper has explored some of the attributes of NMR computer systems. Many features of software voters have been explored both experimentally and theoretically. Section 2 has presented some software voting concepts. N-modular redundancy has been described and the software concepts of time skew and space redundancy have been explained. Various synchronization issues have been presented, including time-outs, points of synchronization, and asynchronous versus synchronous systems. The frequency of voting and the data granularity were shown to be important factors in determining the reliability of NMR systems. Finally, a technique was described to allow easy bit-by-bit voting on words of data.

In Section 4, some experiments were presented to help measure the overhead involved in software voting. The type of voter, the voting frequency, and the average instruction execution time were incorporated into a model of voting overhead. The model was shown to accurately describe the experimental data and an analysis of the programs yielded upper and lower bounds on the possible overhead. The voting frequency was shown to be the dominant factor in determining the voting overhead.

Section 5 shows a number of synchronization experiments. The amount of variation in process execution speed that can be tolerated was determined for three different types of variation. The length of one voter's queue was measured over time to determine how far a process can fall behind two other processes. The queue length was shown to have a bound even when one process is continually slower than the other processes. Guidelines for designing reliable NMR systems were presented, based on the experimental results. A queuing model was developed to describe the length of a subtask's queue over time for any amount of variation in process execution rate. The model was shown to accurately predict the experiments over a range of values.

Many of the ideas mentioned in this article could be developed more fully. The reliability of asynchronous versus synchronous systems could be explored, and the concept of time skew redundancy could be the basis for reliability studies. The assertion was made that as the voting frequency increases there is a point at which the reliability of the system will decrease. This seems intuitive, yet could probably be proven experimentally or mathematically.

I. Voter Algorithms

I.1. Simple Synchronizing Voter

```
Initialize
Loop forever
    For i = 1 to N
        Receive msg i
        Classify msg
        If majority found then
            send msg
    end For
    Report errors
end Loop
```

I.2. Optimized Voter with internal queue

```
Initialize
Loop forever
    Conditional Receive next msg
    If voter buffer full then
        attempt to receive missing msgs
        If majority received then
            vote
        report errors
        initialize oldest msg slot
    store msg
    If majority arrived & not majority found then
        vote
    If all msgs arrived then
        report
        initialize msg slot
end Loop
```

I.3. Sequence Number Voter

```
Initialize
Loop forever
    Conditional Receive
    If illegal sequence number then
        report(" illegal sequence number")
    Else
        Search for seq num in queue
        If seq num found then
            If subtask already sent this seq num then
                report(" seq num duplicated")
            Else
                store msg
                If majority received then
                    vote
                If all msgs arrived then
                    If set no oldest then
                        report(" complete set not oldest")
                    Else
                        report
                        initialize
        Else if seq num not found then
            If queue full then
                handle oldest msg
            store msg in new queue slot
  end Loop
```

References

[1] Davies, D. and Wakerly, J.
Synchronization and Matching in Redundant Systems.
IEEE Transactions on Computers C-27(6):531-539, June, 1978.

[2] Forman, P. and Moses, K.
SIFT: Multiprocessor Architecture for Software Implemented Fault Tolerance Flight Control and Avionics Computers.
Third Digital Avionics Systems Conference :325-329, November, 1979.

[3] Goldberg, J., Weinstock, C., Green, M., Kautz, W., Lamport, L., Melliar-Smith, P.
Development and Evaluation of a SIFT Computer: SIFT Operating System.
Interim Technical Report 2, SRI International, April, 1980.

[4] Goldberg, J.
The SIFT Computer and Its Development.
Fourth Digital Avionics Systems Conference , November, 1981.
SRI International.

[5] Jones, A., and Gehringer, E.
The Cm Multiprocessor Project: A Research Review.*
Technical Report CMU-CS-80-131, Carnegie-Mellon Univerisity, July, 1980.

[6] Kuehn, R.
Computer Redundancy: Design, Performance, and Future.
IEEE Tranactions on Reliability R-18(1):3-11, February, 1969.

[7] McConnel, S., and Siewiorek, D.
CMU Voter Chip.
Technical Report CMU-CS-80-107, Carnegie-Mellon University, March, 1980.

[8] McConnel, S., and Siewiorek, D.
Synchronization and Voting.
IEEE Transactions on Computers C-30(2):161-164, February, 1981.

[9] Michalopoulos, D.
Uniquely Maneuverable Fighter Plane to Use Digital Processors.
Computer , October, 1982.

[10] Ousterhout, J., Scelza, D., and Sindhu, P.
Medusa: An Experiment in Distributed Operating System Structure.
Communications of the ACM 23(2):92-105, February, 1980.

[11] Siewiorek, D., Kini, Mashburn, McConnel, S., and Tsao, M.
A Case Study of C.mmp, Cm*, and C.vmp: Part 1 - Experiences with Fault Tolerance in Multiprocessor Systems.
Proceedings of the IEEE 66(10):1178-1199, October, 1978.

[12] Sindhu, P. and Singh, A.
Performance Evaluation of Message Mechanisms.
Technical Report, Carnegie-Mellon University, 1983.
Computer Science Dept., not yet published.

[13] Singh, A.
Pegasus: A Controllable, Interactive, Workload Generator for Multiprocessors.
Master's thesis, Carnegie-Mellon University, December, 1981.

[14] Snyder, F. G.
A Comparison of Redundant Computer Configurations.
In *Proceedings of Compcon*, pages 125-133. IEEE Computer Society, 1980.
Spring.

[15] von Neumann, J.
Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components.
In *Automata Studies*, , pages 43-98. Princeton University Press, 1956.

[16] Wakerly, J. H.
Reliability of Microcomputer Systems Using Triple Modular Redundancy.
In *Proceedings of Spring Compcon*, pages 23-26. IEEE Computer Society, 1976.

[17] Wensley, J. H.
Industrial-control System Does Things in Threes for Safety.
Electronics 56(2):98-102, January 27, 1983.